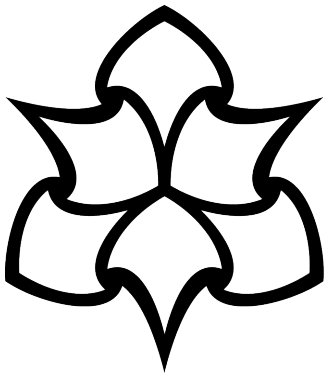

Formulating Efficient Software Solution for Digital Image Processing System

Journal:	*Software: Practice and Experience*
Manuscript ID:	Draft
Wiley - Manuscript type:	Research Article
Date Submitted by the Author:	n/a
Complete List of Authors:	Sherwood, Thomas; Manchester Metropolitan University, School of Computing, Mathematics and Digital Technology Ahmad, Ezak; Manchester Metropolitan University, School of Computing, Mathematics and Digital Technology Yap, Moi Hoon; Manchester Metropolitan University, School of Computing, Mathematics and Digital Technology
Keywords:	software solution, image processing, plug in, maintainability, OpenCV, Matlab

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **00**:1–25
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Formulating Efficient Software Solution for Digital Image Processing System

Thomas Sherwood, Ezak Ahmad, Moi Hoon Yap*

School of Computing, Mathematics and Digital Technology, Manchester Metropolitan University, Chester Street, Manchester, M1 5GD, UK

SUMMARY

Digital Image Processing Systems are complex, being usually composed of different computer vision libraries. Algorithm implementations cannot be directly used in conjunction with other algorithms developed using other computer vision libraries. This paper formulate a software solution by proposing a processor with the capability of handling different types of image processing algorithms, which allow the end-users to install new image processing algorithms from any library. This approach has other functionalities like capability to process one or more images; manage multiple processing jobs simulteneously; and maintain the manner in which an image was processed for later use. It is a computational efficient and promising technique to handle variety image processing algorithms. To promote the reusability and adaptation of the package for new types of analysis, a feature of sustainability is established. The system past the testing procedures by using unit testing, integration testing and usability testing. Future work involves introducing the capability to connect to another instance of processing service with better performance. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: image processing, software solution, sustainability, plugin, OpenCV, Matlab

1. INTRODUCTION

Digital image processing is complex and inconsistence due to various programming languages and variation computer vision libraries. The domain of image processing has increased vastly in recent years [1], spanning across a range of applications such as photography, forensics and medical imaging [2]. The term simply relates to the process (or set of processes) applied to the detector and dataset of a radiograph exposure [3]. Motivations for processing an image stem from not only the amount of information perceived as image form, but also for autonomous machine control [4].

A mechanism for implementing the algorithms is required, in order to provide a means to perform the transformations. Larkins et. al. [5] discuss an existing high-level toolbox known as Matlab, providing a plethora of existing algorithms and components for re-use in building more complex algorithms. They highlight how Matlab is easy enough for novice users to grasp while still providing powerful processing and data crunching capabilities. However algorithm implementations cannot be directly used in conjunction with other algorithms developed using other technologies, for instance C++ processing classes. Culjak et. al.[6] discusses an alternative to Matlab known as OpenCV, which provides a suite of processing algorithms and assistant classes written in C. They discuss how the library is also widely used, providing heavily optimised solutions to particular algorithms. A C++ wrapper is available for OpenCV, allowing for easy integration into higher-level languages.

*Correspondence to: School of Computing, Mathematics and Digital Technology, Manchester Metropolitan University, Chester Street, Manchester, M1 5GD, UK.

Matuska et. al.[7] provides a detailed comparison between the processing speeds of Matlab and OpenCV, concluding that OpenCV dominates Matlab in regards to processing speed. However in order to utilise OpenCV, the user must directly implement the processing code in a language capable of using the C or C++ implementations.

The contribution of this paper is to formulate a Digital Image Processing System (DIPS), by proposing a processor with capability of handling different types of image processing algorithms. The functionality requirements include: capability to process one or more images; allow end-users to install new image-processing algorithms; manage multiple processing jobs simultaneously; and maintain the manner in which an image was processed for later additional analyses. In addition to these functional requirements, there are non-functional requirements for this project; i.e. memory and computational efficiency. The sustainability of DIPS is also an issue to be considered. It has been established that a feature of sustainability is that a system is not simply being made bespoke and obsolete as soon as its initial use concludes [8]. This is ensured by the design to keep the new image processing algorithms created by the user that promote reusability and adaptation of the package for new types of analysis.

This paper is organised as follows. Section 2 presents the concept of DIPS and the detailed design. Section 3 shows how to implement the processing modules in DIPS. Section 4 describes the testing process and Section 5 conclude the paper with future work.

2. PROCESSOR DESIGN

This section pertains to the design of the image processing module within the DIPS application. Opening with a high-level view of the communication between the application and the processing modules, the core requirements of the processor are devised and explained in detail. These requirements are used as the base of the system design description.

2.1. System Architecture

The formulation of DIPS application is composed of three key components: the database system used for persistence, the graphical user interface providing the presentation layer, and the image processor module. The processor discussed in this paper will permit any image which can be represented as an object. This provide the flexibility of supporting different image formats.

The processing module is intended to run independent of the DIPS application as a service. This allows the deployment of the processor and execution of jobs in a separate environment (and optionally machine) to a running DIPS instance, the intent of which paves the way for the potential for shared ‘cloud’ computation. The functions of the processor are exposed through a public interface known to the application, which prevents locking it to a single instance of the service and provides the means to test the application through mock instances.

The application-layer is responsible for manufacturing an object describing the job to be executed. This consists of the inputs to process in addition to the method of processing each image. The processor uses this information to execute the processing job, firing events back to the client (such as when work begins and ends, when a single image is processed and so forth). The processor is broken down into several subsystems responsible for various operations. One such module provides job-management functionality, allowing clients to submit more than one job and leave the processor to complete them in the order provided. On receiving a new job, the processor enqueues the information into the job queue and provides the caller with a ‘ticket’ representing the job within the queue. The application will later use this ticket to access their results, or use it to observe events as the job progresses. The new job is enqueued into the job queue, which maintains the current set of active jobs to process. The queue will then fire an event notifying observers a new item has been added.

A separate subsystem to the job management system provides the actual execution of jobs within the processor. On observing the event fired by the queue, a background worker initialises (if it is not already working). This worker dequeues the next job and uses the information to execute the processing functionality, while notifying the client as it progresses through the provided inputs. If

DOI: 10.1002/spe

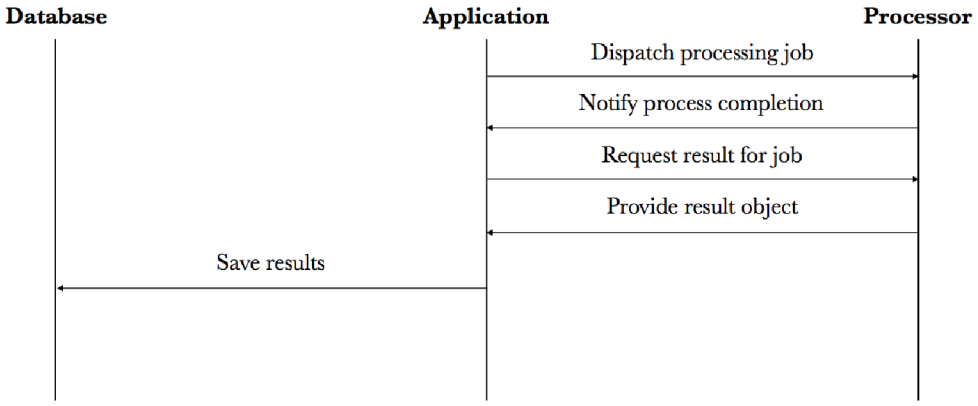

Figure 1. High-level communication between application and processor

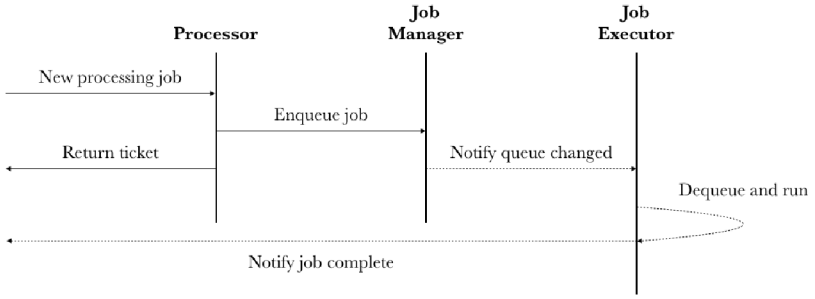

Figure 2. New jobs are provided to the job manager, which creates the executable job based on the information provided by the client. It is then enqueued and ran at a later time. Many jobs can be queued as the background worker gradually completes them in a FIFO manner. Dashed lines represent asynchronous actions.

an error occurs while running a job, it notifies the client of the error and aborts the execution of that particular job. The next job is then dequeued and ran, until the queue is exhausted. Another subsystem provides the means to convert the information provided by the client into the actual job. The client is provided with objects used to represent the processing algorithms, which must be converted into their associated instances before they can be used. This prevents the introduction of dependencies into the client, and instead only exposes limited types through the public interface. Additionally, new image processes can be installed into the service and sent to the client in these 'definition' objects without needing to alter the client or the underlying processor interfaces and logic.

Finally, the output of processing each image is retained locally by the processor until the client requests the results. They can then use the ticket to delete the results once they have accessed them, or leave the service to clean up outstanding results on shutdown. In the DIPS application, the results are retrieved and saved to the database.

2.2. *Requirements*

The processing service used by the application is subject to several functional requirements, either inherited from the main goals of the application or derived based on feedback by stakeholders. These include:

- *Provide the capability to process one or more images.* While an obvious requirement, the processor must expose the capability to accept one or more images in a request and provide the means to process them using an arbitrary algorithm chosen by the client. This is to decouple all image processing logic from any application logic.

DOI: 10.1002/spe

- *Allow end-users to install new processing algorithms.* Rather than provide a limited set of algorithms and be done with it, there is the potential an end-user may want to incorporate a specific image processing technique they have written into the processor. Rather than have to receive and modify a copy of the processor, they should be able to incorporate their processing algorithm without needing to make other changes.

- *Manage multiple processing jobs simultaneously.* The application can enqueue dozens of jobs which are completed sequentially, with the client receiving the results as each job completes.

- *Persist the manner in which an image was processed for later re-use.* The main objective of the DIPS application is to save the end-user time when performing pre-processing operations. While this is achieved by batch processing, they may wish to re-use the same set of processes at a later point in time (for instance, as new scans become available). To satisfy this, the processor needs to be able to save the state of the method used to process images and provide the capability to use this information at a later time to restore the method's state.

In addition to the functional requirements listed above, there are several non-functional requirements requiring consideration when implementing the processing module. These include:

- *Memory.* As the processor will be dealing with a large number of requests simultaneously, all composed of multiple image files, the memory required for the processor to run will increase greatly based on the size and number of processing batches it has received. A degree of memory management may be required if the processor begins occupying too much memory during testing.

- *Computational Efficiency.* The amount of processing required to execute a job is heavily dependant on the way in which they wish to process their inputs. If they have chosen a processing mechanism which takes a considerable amount of time, or have requested a very large set of images to be processed, the time their job will take will increase and cause other queued jobs to be delayed further. The processor must decide if a job should be aborted for exceeding the allowed time permitted by the job to complete, whether or not this is based on the complexity of the algorithm or another reason for the blocking call.

2.3. Subsystem Architectures

2.3.1. Process Definitions Job requests dispatched to the processor are composed of not only the inputs to be processed, but also an object specifying how to process the images. In order to construct a legal processing definition, the application must first be aware of the type of object used to define a process in addition to receiving objects of this type depicting the available processes. The client may also wish to use a complex process consisting of one or more other processes, effectively chaining them together.

This chaining of processes together is regarded as a 'pipeline'. The pipeline consists of one or more elements representing an image process, in which the image is passed through the individual elements until it exits the pipeline. Each element is capable of accepting an arbitrary image input, perform the processing task it represents, and output the result for use by the next element in the pipeline. This isolates each process from one another by eradicating knowledge of other processes from one another, and additionally allows for greater customisation by the user; rather than hard-coding an algorithm or only allowing the selection of a single process, users are able to replace components of a pipeline, or add or remove elements as their processing needs dictate. The objects exposed to the client for the purposes of pipeline construction are relatively basic in nature in order to avoid introducing too many dependencies. The *AlgorithmDefinition* object represents a single processing algorithm available to the processor, consisting of its identifier and a separate object containing its parameters. These are aggregated within a *PipelineDefinition* object, assembled by the client to represent their desired processing task. Figure 4 depicts this relationship between the objects. The processor exposes a read-only collection of available *AlgorithmDefinition* objects which the client can utilise to build legal pipelines. These are resolved when the processor

DOI: 10.1002/spe

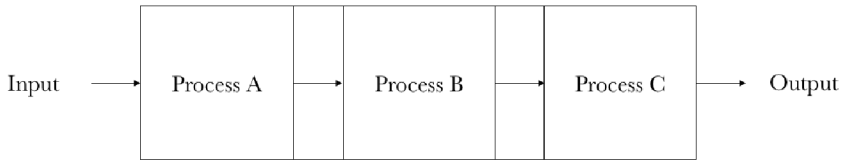

Figure 3. The processing algorithm packaged as part of a job request is in fact a 'pipeline' composed of one or more image processes

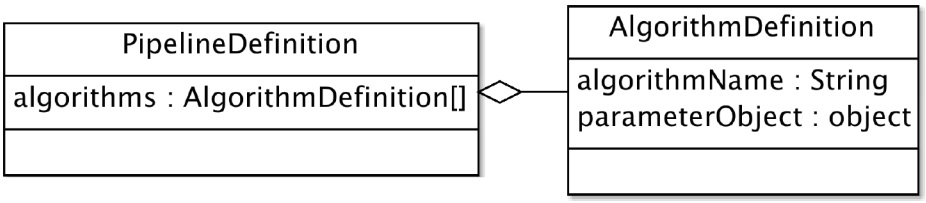

Figure 4. Client applications use AlgorithmDefinition objects alongside PipelineDefinitions to portray their desired processing pipelines

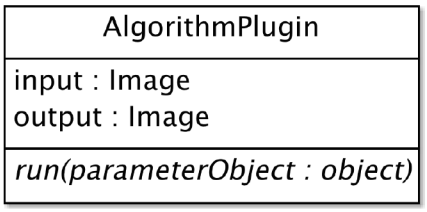

Figure 5. The abstract *AlgorithmPlugin* represents a single image processing technique. The processor utilises the common interface to communicate with implementations without requiring them to possess knowledge of each other.

first initialises and are not modified during the lifetime of the processor. The range of available processing techniques available to the client is dependant on the implementation of the processor. It retains the underlying implementations privately and exposes the *AlgorithmDefinitions* publicly. This separates how the implementations of the processes are dealt with from the client, allowing for restricting of the internals of the processor without breaking existing pipelines in use by clients.

In the case of the DIPS design, it is required that new processing techniques can be implemented and adapted into the processor without requiring the modification of any existing code. To achieve this, an architecture based around the concept of 'plugins' is required. These plugins represent unique image processing techniques, such as histogram equalisation or gamma correction. The processor is made aware of the existence of these plugins and loads them dynamically, exposing them as *AlgorithmDefinition*s to the client through the public interface.

To achieve this, a base class is required in order for the processor to communicate with a plugin. The *AlgorithmPlugin* class (figure 5) represents the basic definition of an image processing technique; it utilises a known *Image* class representing the input, and performs its processing implementation through a call to its *run* method. This method accepts an arbitrary object representing the parameters to the process, provided by the client through the associated *AlgorithmDefinition* object. The processor is able to instantiate an instance of this class by associating the specific plugin implementation with the identifier used in it's *AlgorithmDefinition*. Upon receiving a new job request from the client, the processor uses the *PipelineDefinition* to reconstruct the collection of associated *AlgorithmPlugin* objects. It sets the input of each plugin as the output of the former (or in the case of the first process, the input image itself). The output of the final process is then regarded as the output of the pipeline. This approach of dynamically loading plugins and exposing them as *AlgorithmDefinition*s allows the development and installation

DOI: 10.1002/spe

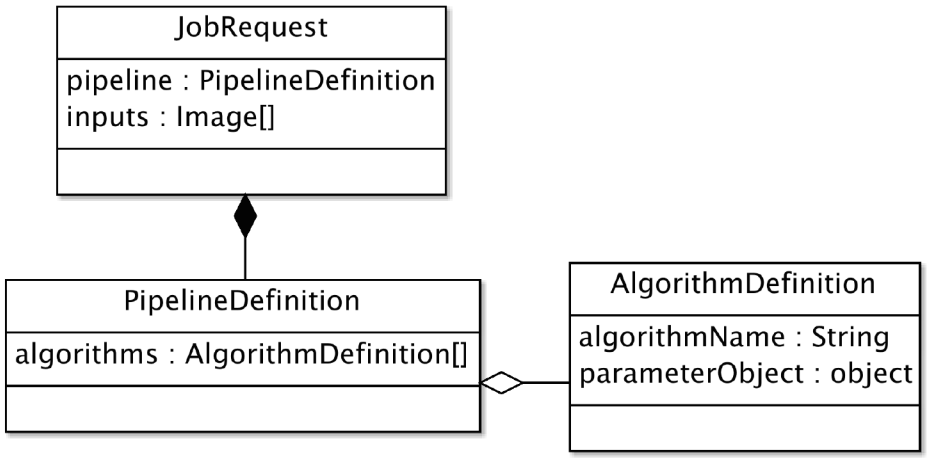

Figure 6. The client constructs a *PipelineDefinition* as part of their *JobRequest*, along with the set of input *Image*s to be processed. The processor uses this information to constructing processing jobs.

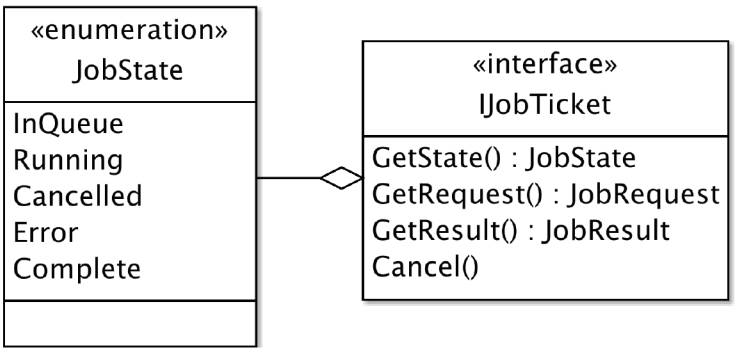

Figure 7. The client is provided with a job ticket, representing the job within the processing queue. The client can attach to relevant events from this object, or even cancel the job if required

of new processes without modifying either processor or application code, while keeping the system testable by isolating all the components from each other.

2.3.2. Job Management Given a mechanism of allowing clients to define tasks through *JobRequest*s, the processor must be capable of transforming these requests into real jobs it can process. This is undertaken using a queueing system in which jobs dispatched by the client application are executed in a FIFO manner. Given a task outlined by a client's *JobRequest*, the processor must first validate the contents of the job. The *PipelineDefinition* provided is examined, and made sure it does not contain any *AlgorithmDefinition*s the processor is unaware of. Upon successful validation, the information is retained by the processor and the client is provided with a 'ticket'. Like a real queueing system, the ticket represents the jobs place within the queue and can be used at later points within the job's lifetime to perform actions against it. The processor retains the ticket also, used by a separate object to devise and execute the job it represents. On dispatching the first job to the processor, the queue signals an event declaring it contains pending jobs. A background worker observes this event, and begins the execution loop. The background worker dequeues the next ticket from the queue and begins executing the job. It uses the ticket to gather the information provided by the client in the *JobRequest* and dispatch appropriate events as it makes progress. This is executed on a separate thread, providing a producer-consumer relationship between the two objects. As clients continue feeding the processor with jobs, the worker will gradually execute the work in the background until the queue is exhausted.

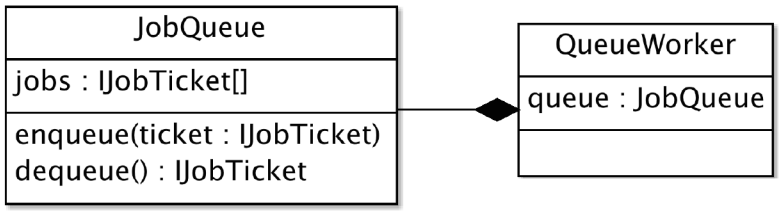

Figure 8. The *JobQueue* retains the FIFO data structure collating the jobs received by client applications. The *QueueWorker* sequentially dequeues and works with the next job in the queue.

```
<?xml version="1.0" encoding="utf-8"?>
 <pipeline>
  <algorithms>
   <algorithm name="GammaCorrection">
    <properties>
     <property name="Gamma" value="3"/>
    </properties>
   </algorithm>
  </algorithms>
 </pipeline>
```

Figure 9. Pipelines created by applications can be represented as XML and re-loaded at a later time. The processor can convert this XML back into its represented *PipelineDefinition* by re-instantiating the appropriate *AlgorithmPlugin*s and providing them with their associated XML.

2.3.3. Pipeline Persistence One of the key requirements of the processor is to save time in the recreation of process batches by saving the users processing workflow. While they can process a large batch of images using a single pipeline, it is likely they will want to retain the pipeline they have created for reuse at a later time. Thusly a mechanism is needed to retain the state of their customised process to a data store. The *AlgorithmDefinition* objects used to expose the pipeline components carry enough state information to represent a process. They state which process in particular they represent in addition to possessing an object detailing the parameters to be used when the process is executed. However, these objects may not be safe to serialise into a permanent form into a data store - it is never known what contents the parameter object may contain, nor is it safe to assume the object itself will change. Instead of object serialisation, XML can be used to represent the individual processes of a pipeline (figure 9). The *AlgorithmPlugin* objects loaded and used by the processor can expose the capability to create or load XML in order to create a memento of their state. In the event their schema changes (due to the addition of a new property or otherwise), they can act accordingly against older versions of their XML. This is much safer than attempting to reload stale serialised objects, and also allows the testing of the pipeline system without any UI; XML representing pipelines can be injected to create the relevant pipeline without needing a UI to construct it. The processor provides the two-way pipeline to XML transformation on the public interface. Once a client has constructed a pipeline, they can hand the *PipelineDefinition* object to the processor. The associated plugins are then used to create their individual *algorithm* elements represented in the DOM in figure 9. These are then used to form the document itself, which is returned back to the client. It is up to the application how to store the XML, whether in the file system or a database. On receiving XML to re-create a *PipelineDefinition*, the processor resolves the appropriate plugin using the *name* attribute in each *algorithm* element. It then hands the plugin the particular element to restore itself, before converting them into their associated *AlgorithmDefinition* objects. In the event invalid XML is provided, the processor throws an appropriate exception for the client to handle.

DOI: 10.1002/spe

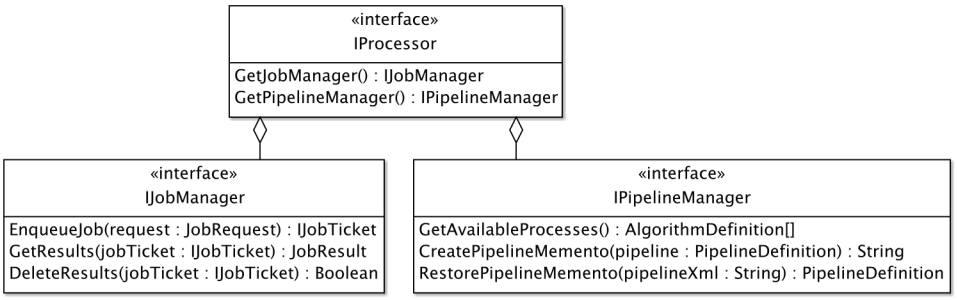

Figure 10. The processor module is represented by the *IProcessor* interface, exposing further submodules providing pipeline and job management.

2.4. Client Interface

Previous sections have discussed how the requirements of the processing module have been satisfied within their subsystems, providing the means for client applications to execute the required functions. With these subsystems in place, the manner in which they are exposed can be devised. The client application is intended to obtain a reference to a root interface it is aware of, which represents the processing module. This interface is further composed of the submodules discussed previously, pertaining to job management and pipeline building/persistence. Figure 10 represents this structure, and an overview of the general methods applicable to each component.

IProcessor

The root *IProcessor* is intended to represent the entirety of the processing module. It is an instance of this interface in which client applications are provided with, granting them with the capabilities represented by the processor. At present, it does little more than expose the associated subsystems providing the underlying functionalities of the processor.

IJobManager

The *IJobManager* is intended to represent the subsystem used by client applications in which jobs are dispatched to. Clients can call the *EnqueueJob* method with their *JobRequest* to enqueue their processing job, providing them with the *IJobTicket* representing their place in the queue. They can listen to events on the ticket and await completion of their work, before calling the *GetResults* method on the manager. This provides them with a *JobResult* object containing the output of their processing task. They can then optionally call *DeleteResults* once they have retained a local copy to tidy up their work on the processor immediately. Clients who are neglectful in this regard will have their results automatically removed after an extended period of time, or when the object is disposed.

IPipelineManager

The *IPipelineManager* provides clients with the means to become aware of the available *AlgorithmDefinition*s within the associated *IProcessor*. It exposes these definitions through a simple getter, allowing the application to then build up their *PipelineDefinition*. The client will then use this in the *IJobManager* as part of their *JobRequest*, however it is the *IPipelineManager* who exposes the functionality to save and restore *PipelineDefinition*s to and from Xml. Two methods exposing these functionalities are available against the manager, accessible to the client.

2.5. Summary

The individual components described within this chapter provide the processing functionality required by the DIPS application. However, the design approach taken does not restrict the processing capabilities to one particular implementation of the software and is instead made available to any application aware of its client interface. A discussion of the mechanism providing

DOI: 10.1002/spe

the processes was provided, detailing the usage of a plugin framework capable of loading the available processing algorithms dynamically. This prevents restricting the capabilities of the processor to a particular subset of processing algorithms during the current development cycle, and instead provides scope for other developers to continue adding their own processing algorithms at a later point in the product lifetime. The manner in which jobs are managed within the system was also discussed, detailing the requirement of a robust queueing system. This is used in conjunction with multithreading to allow a constant throughput of work provided the processor continues receiving work. Following this, the manner in which processing 'pipelines' are persisted was devised. The chosen manner of performing this was Xml, as it avoids the pitfalls of object deserialisation. Additionally, it provides the means to design a pipeline without requiring a UI or additional code to inject a pipeline definition. Finally, the manner in which the core functions of the processor are exposed to the public was presented. Separating the logical subsystems of the processor into separate domains makes it easier to manager for both the client and the processor implementation.

3. IMPLEMENTATION

This section of the document details the specifics of the implementation of the processing module within the DIPS application. It has been broken down into logical segments pertaining to relevant sections within the design, such as the plugin system and job management.

3.1. Plugin System

This section focuses on the plugin subsystem and its interactivity with the rest of the processing module. A discussion of the system was provided along with a high-level overview of its interactions (page 5, figure 5). There are however further steps required for consideration:

1. How implementations of plugins are loaded into the processor dynamically
2. How the parameter object associated with the process is resolved and created
3. How the two-way Xml procedure is implemented to provide a means of process persistence

3.1.1. Dynamic Plugin Loading The ability to load new plugin implementations into the processor without requiring the modification of existing code is one of the key features of the design. Other developers can implement new processing algorithms and incorporate them into pipelines without modifying the processor itself. The manner in which plugins are integrated is relatively straightforward, making use of a concept known as type introspection. This is where the program has the capability to examine and potentially modify itself during runtime, however we only make use of the former. The C# language makes this available through *Reflection*, which is built right into the language. It allows the analysis of class structures as an object of type *Type*, exposing details such as property names and types, attributed values, superclasses, interfaces and more. Reflection can also be used to examine compiled assemblies, also regarded as DLL files. Assemblies represent an aggregation of types and other code, in which reflection provides enumeration over the types within the assembly. Figure 11 demonstrates the loading of an assembly from a file, before enumerating through all the *Type*s within the assembly. The *Type* class provides the means to determine whether the class it represents extends another class. Additionally, the *typeof* operator allows the introspection of a class without requiring an instance of it. Thusly, we can combine the two as demonstrated within figure 12 to determine whether a class represented by a *Type* extends the *AlgorithmPlugin* class. Combining the code snippets in figures 11 and 12, the implementations of *AlgorithmPlugin* within a particular assembly can be uncovered. However, this only yields their *Type*s and does not associate them with any sort of identifier. At this stage of the process, adding an abstract accessor requesting their identifier will not solve this as the types have yet to be instantiated. Therefore an attribute must be required to be applied to the class in order to resolve this. In C#, an attribute is a class which extends the appropriately named *Attribute* class. Such classes are

DOI: 10.1002/spe

```
Assembly assembly = Assembly.LoadFrom( "MyAssembly.dll" );
foreach( Type type in assembly.GetTypes() )
{
   // Do stuff with type
}
```

Figure 11. Once an assembly has been loaded, the *Type*s within the assembly can be enumerated through. Each *Type* represents a class, interface or other entity and can be introspected further.

```
if( type.IsSubclassOf( typeof( AlgorithmPlugin ) ) )
{
   // Type subclasses AlgorithmPlugin
}
```

Figure 12. The *Type* class provides the *IsSubclassOf* method to determine whether the class it represents extends another, represent by it's *Type*

```
[AttributeUsage( AttributeTargets.Class, AllowMultiple = false )]
public sealed class AlgorithmAttribute : Attribute
{
   public AlgorithmAttribute( string pluginName )
   {
      PluginName = pluginName;
   }

   public string PluginName
   {
      get;
      private set;
   }
}

[Algorithm( "SomeAlgorithm" )]
public class SomeAlgorithm : AlgorithmPlugin
{
   ...
}
```

Figure 13. The *AlgorithmAttribute* provides the means to 'annotate' classes with an identifier. This can then be used to associate an identifier with a type of plugin, rather than a single instance.

used to 'annotate' various parts of code files, such as classes, properties, methods and so on. In this scenario, each *AlgorithmPlugin* implementation must be annotated with a class requesting them to provide an identifier. Figure 13 demonstrates an implementation and usage of this class, providing a specific *AlgorithmPlugin* with an identifier within its type definition. The *Attribute*s of a *Type* can be accessed with ease through a simple method call. If no attribute is found then the plugin is ignored, otherwise the identifier and type can be registered into the system. By checking whether the class represented by *Type* extends *AlgorithmPlugin* and has been annotated with the *AlgorithmAttribute*, plugin implementations can be loaded from DLLs provided the processor. In the case of the DIPS solution, the assemblies it is provided with reside in the same directory as the executing processor assembly. On startup, the processor checks the directory it is executing in for

DOI: 10.1002/spe

```
var attribute = type.GetCustomAttribute( typeof(
   AlgorithmAttribute ) );
if( attribute != null )
{
   // Plugin has AlgorithmAttribute
}
```

Figure 14. Using introspection, the *AlgorithmAttribute* can be resolved from a *Type*. If the *GetCustomAttribute* method returns null, the implementation has not been annotated with the attribute and the plugin is ignored.

```
string currentDir = Directory.GetCurrentDirectory();
var files = from file in Directory.GetFiles( currentDir )
          let fileNoPath = Path.GetFileName( file )
          where fileNoPath.StartsWith( "DIPS.Processor.Plugin" )
          && fileNoPath.EndsWith( ".dll" )
          select fileNoPath;
foreach( string fileName in files )
{
   // Assembly is a plugin assembly
}
```

Figure 15. Plugin assemblies within the current directory can easily be located by enumerating over the files and filtering out those not named like 'DIPS.Processor.Plugin.*.dll'.

compiled assemblies and scans each one for plugin types. When a new plugin is to be integrated into the processor, it need only be compiled into a DLL and copied into the same directory as the processors binaries. This manner of loading plugins through DLLs is very flexible, only requiring future developers to extend the base *AlgorithmPlugin* class, attribute the class accordingly and copy the compiled output to the same location as the processor binaries. Figure 16 demonstrates the entirety of the plugin loading procedure, from detecting assembly files to scrutinising types within assemblies. Error catching has been omitted for clarity. When the processor is required to convert the information registered about the plugin into an actual object, reflection can be used again to instantiate the associated *Type* into the implementation of the *AlgorithmPlugin*. This is demonstrated in figure 17, through a call to a method against the *Activator* static class. With the ability to load the plugins from their assemblies, a mechanism is required for persisting this information during the lifetime of the processor. This is retained in a 'registrar' held by the processor itself, which can be provided to other components requiring plugin-level functionality. The *IAlgorithmRegistrar* represents the container for this information, in which a specific implementation is held by the *IProcessor*. The *IAlgorithmRegistrar* retains the information about all loaded plugins, but does not possess the capability to actually manufacture them. This behaviour resides within a separate factory utilising this information known as the *IPluginFactory* (figure 19), which provides the ability to manufacture an *AlgorithmPlugin* from its *AlgorithmDefinition* by using the information within the registrar. This separation of concerns is not only to ensure these distinct behaviours are isolated, but it also helps keep the system testable. Specific tests can be written to simulate scenarios within the *PluginRegistrarFactory* by using dependency injection. Various implementations of the *IAlgorithmRegistrar* can be provided to simulate lack of knowledge of particular algorithms amongst other conditions.

3.1.2. Plugin Parameter Objects As previously stated in the design, implementations of *AlgorithmPlugin* provide the ability to perform the image processing algorithm against an input. The parameters used to adjust their execution pattern reside in a separate object, providing during their

DOI: 10.1002/spe

```
private void _loadAssembliesInCurrentDirectory()
{
   string currentDir = Directory.GetCurrentDirectory();
   var files = from file in Directory.GetFiles( currentDir )
               let fileNoPath = Path.GetFileName( file )
               where fileNoPath.StartsWith( "DIPS.Processor.Plugin"
                   )
               && fileNoPath.EndsWith( ".dll" )
               select fileNoPath;
   foreach( string fileName in files )
   {
      _loadssemblyByName( fileName );
   }
}

private void _loadAssemblyByName( string assemblyFileName )
{
   Assembly assembly = Assembly.LoadFrom( assemblyFileName );
   foreach( Type type in assembly.GetTypes() )
   {
      _examineType( type );
   }
}

private void _examineType( Type type )
{
   if( type.IsSubclassOf( typeof( AlgorithmPlugin ) ) )
   {
      var attr = type.GetCustomAttribute( typeof(
         AlgorithmAttribute ) );
      if( attr != null )
      {
         // Plugin detected
      }
   }
}
```

Figure 16. When the processor is first created, it calls *loadAssembliesInCurrentDirectory* to register any plugins it can find. The assembly is then loaded in *loadAssemblyByName*, which in turn begins scrutinising each detected *Type*. If the *Type* represents a plugin, it is registered into the plugin system.

```
Type pluginType = ...
AlgorithmPlugin plugin =
(AlgorithmPlugin)Activator. CreateInstance(pluginType);
```

Figure 17. The *Activator* type provides the means to instantiate the actual object represented by their *Type* counterpart. The processor assumes the plugin implementation contains a parameterless constructor for this to work; the processor cannot be aware of the construction requirements of a plugin without needing to be aware of their implementation.

definition. However *AlgorithmDefinition*s do not utilise an instance of the plugin, and thus a factory

DOI: 10.1002/spe

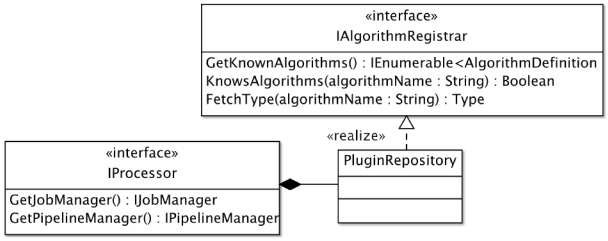

Figure 18. The *IProcessor* possesses an *IAlgorithmRegistrar* implementation, capable of providing information about loaded plugins and resolving the *Type* associated with their identifier.

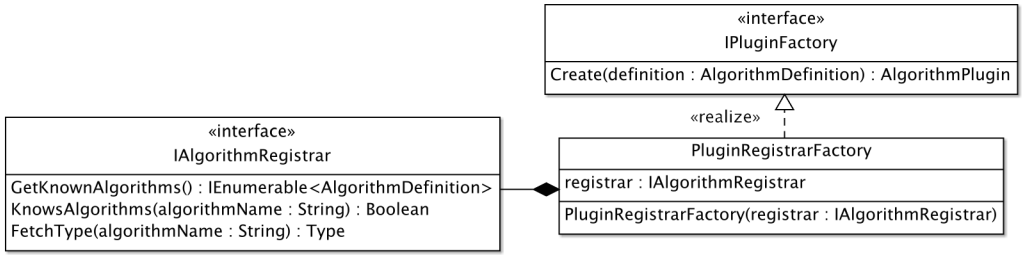

Figure 19. The *IPipelineFactory* uses the information provided by the *IAlgorithmRegistrar* to manufacture plugin instances using their associated *AlgorithmDefinition*s.

method on the plugin to create an instance of their parameter object will not work. The solution to this problem is simple, returning to the *AlgorithmAttribute* class detailed in figure 13. At present, the class only requires implementations to provide an identifier as part of their metadata. We can amend this to include the *Type* of the parameter object used by the plugin for later re-instantiation. As plugins may not require properties (i.e., they perform one strict path of logic) this additional property is left optional. C# supports this, as depicted in figure 20, in which optional parameters are given values by name. Additionally, a restriction is set against the *Type* to ensure it implements a common *ICloneable* interface. This exposes a singular *Clone* method, which returns a copy of the object in its current state. This prevents the use of reflection to re-instantiate the parameter object every time an *AlgorithmDefinition* needs to be built, allowing the cloning of the initial instance. Besides being a much faster operation, it is also far safer as instantiating objects from their *Type* can be very exception-prone.

3.1.3. Xml Persistence Retaining the state of a *PipelineDefinition* is to be implemented through Xml, as stated in the design. This is to avoid dangerous scenarios pertaining to deserialisation of class data into a different version of the class. It was previously demonstrated how each algorithm will be required to generate and accept its own Xml (figure 9). Rather than expose this functionality as part of the *AlgorithmPlugin* class, logic pertaining to state management of plugins is done through a separate class entirely (figure 21). This class is responsible for accepting previously generated Xml and converting it into an instance of the plugins parameter object, in addition to accepting the parameter object and generating its Xml. Similar to plugins themselves, these memento classes must be annotated with a specific attribute in order to expose the type of *AlgorithmPlugin* they generate Xml for. When plugins are loaded into the processor, their associated memento class is located and loaded into the plugin system. Figure 22 depicts this class. While this implementation requires client plugins to implement an additional interface, it allows them to specify exactly how their plugin's parameter object should be represented as Xml. An alternative implementation is the use of reflection to access the public properties of the parameter object and serialise their values to binary, however this again raises the serialisation of types problem in the event the parameter object is composed of further objects, allowing the potential for deserialisation problems down the line. With the association formed between *AlgorithmPlugin*s and their *IPipelineXmlInterpreter*

```
[Algorithm( "SomeAlgorithm", ParameterObjectType = typeof(
    SomeParameters ) )]
public class SomeAlgorithm : AlgorithmPlugin
{
    ...
}

public Type ParameterObjectType
{
    get
    {
        return _parameterObjectType;
    }
    set
    {
        if( value != null &&
            value.GetInterfaces().Contains( typeof( ICloneable ) )
        {
            _parameterObjectType = value;
        }
    }
}
[DebuggerBrowsable( DebuggerBrowsableState.Never )]
private Type _parameterObjectType;
```

Figure 20. Plugins specify the *Type* of their parameter objects when they annotate themselves with the *AlgorithmAttribute*. This allows the processor to later re-instantiate the type and provide the object to the relevant *AlgorithmDefinition*.

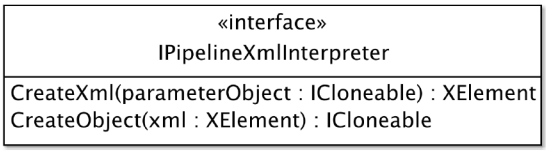

Figure 21. The *IPipelineXmlInterpreter* provides the interface allowing the processor to call memento methods relevant to an *AlgorithmPlugin*.

implementations, Xml documents can be created and loaded providing the pipeline persistence needed. Both operations exposed through the *IPipelineManager* (figure 10).

Creating Xml The calling application can call the *CreatePipelineMemento* method against the *IPipelineManager* in order to create the relevant Xml for the *PipelineDefinition*. As the *AlgorithmDefinitions* it is composed of are associated with a particular *AlgorithmPlugin* known to the processor, its associated *IPipelineXmlInterpreter* can be resolved. Creating the Xml using this information is straightforward, in that most of the execution is handled by the plugin classes. The processor will create a root element for each *AlgorithmDefinition*, providing it with an attribute consisting of the identifier of the plugin. This is to identify the plugin when reloading the Xml at a later date. The processor then calls the *CreateXml* method on the plugins *IPipelineXmlInterpreter*, providing it with the parameter object within the *AlgorithmDefinition*. It then sets this Xml as the first child, and repeats this process for each *AlgorithmDefinition*. The resulting collection of Xml elements is then wrapped into a document, dictated by the schema from the design (figure 9).

DOI: 10.1002/spe

```
public sealed class XmlInterpreterAttribute : Attribute
{
   public XmlInterpreter( Type pluginType )
   {
      if( pluginType == null ||
         pluginType.IsSubclassOf( typeof( AlgorithmPlugin ) ) ==
            false )
      {
         throw new ArgumentException(
            "Type provided to XmlInterpreter must subclass
               AlgorithmPlugin" );
      }

      PluginType = pluginType;
   }

   public Type PluginType
   {
      get;
      private set;
   }
}
```

Figure 22. The *XmlInterpreterAttribute* provides the means to expose *IPipelineXmlInterpreter* implementations to the processor. The *Type* they are provided must represent an *AlgorithmPlugin*, which is then associated with a loaded plugin implementation.

```
[Algorithm( "SomeAlgorithm", ParameterObjectType = typeof(
   SomeProperties )]
public class SomeAlgorithm : AlgorithmPlugin
{
   ...
}

public class SomeProperties : ICloneable
{
   ...
}

[XmlInterpreter( typeof( SomeAlgorithm ) )]
public class SomeXmlInterpreter : IPipelineXmlInterpreter
{
   ...
}
```

Figure 23. Example usage of exposing an *IPipelineXmlInterpreter* that is used to create or restore from Xml against the *SomeAlgorithm* plugin.

Restoring from Xml While creating Xml is a straightforward process, loading from it is slightly trickier. As the usage of existing C# Xml classes is in place, validation that the document's syntax is ensured. However the actual structure of the elements and their contents are subject to scrutiny, as they could easily have been tampered with. This functionality is broken down into more manageable

DOI: 10.1002/spe

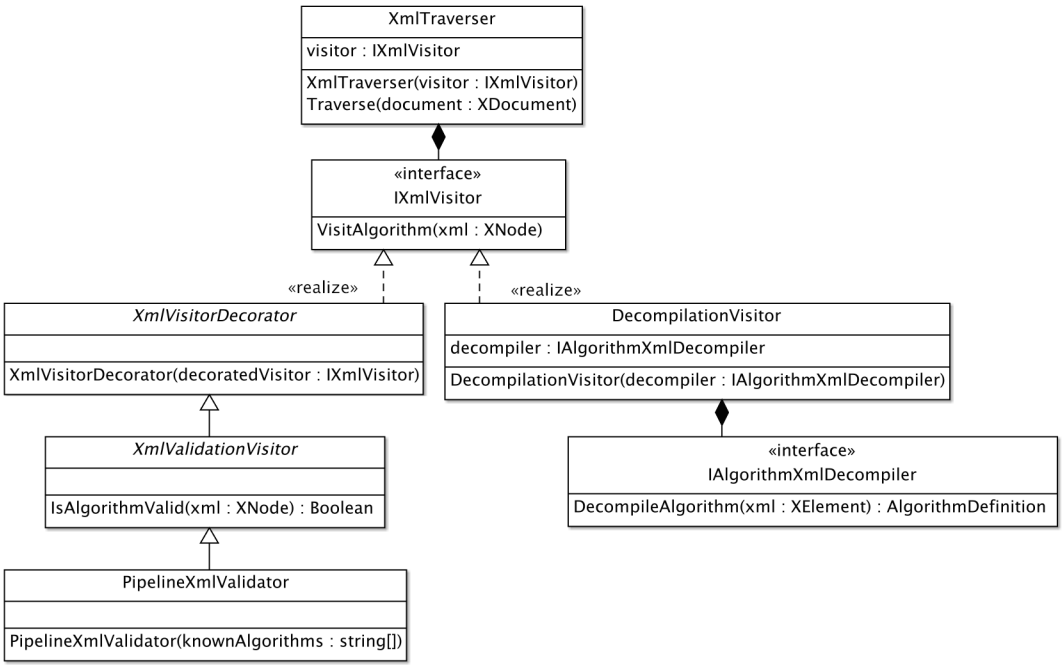

Figure 24. The decorator and visitor patterns greatly broaden the reuse of loading Xml into the processor. The visitor decouples the Xml traversal logic from the logic utilising the Xml. The decorator allows 'compounding' behaviours together.

steps. Given an Xml document, a particular object traverses through its structure and locates the appropriate algorithm elements within the three. It then sends this information to a 'visitor' class used to try and parse the Xml, using the interpreters loaded through the plugin system. Once the traversal procedure is complete, the parsed Xml is then used to form a new *PipelineDefinition*. A distinct layer of validation is required between the traverser and the visitor to ensure the schema is correct. Rather than integrate this into the decompilation logic, a decorator can be used. This provides the means to compound behaviours together dynamically, in this case providing validation logic to decompilation logic. This is demonstrated in figure 24. The *DecompilationVisitor* uses an *IAlgorithmXmlDecompiler* to convert the Xml back into *AlgorithmDefinition* objects. It then retains the definitions for collection later once the traversal is complete. The processor will manually create an *XmlTraverser*, and provide it with the decompiler. It will utilise the *PipelineXmlValidator* in order to decorate the decompilation behaviour with the validation behaviour, ensuring the document is valid while parsing. This greatly separates the different behaviours required for this procedure, allowing for more traversal mechanisms to be implemented while keeping the design testable.

3.2. Job Management

This section focuses on the implementation of the job creation, queueing and execution components of the processor. It also discusses the mechanisms in place for notifying client applications of the progress of their processing jobs.

3.2.1. Job Creation The design discusses how client applications are provided with a limited set of objects detailing the available processing algorithms within the processor (figure 6). The *JobRequest* they provide to the processor is composed of the *PipelineDefinition* they have constructed, along with the set of *Image* objects to be processed. This is provided to the *IJobManager*, which enqueues the job and returns an *IJobTicket* (figure 10). The *IJobManager* does not retain the job information itself, nor any of the queueing functionality (discussed later). It instead delegates to a number of other modules pertaining to converting the *JobRequest* information into a ticket and enqueueing

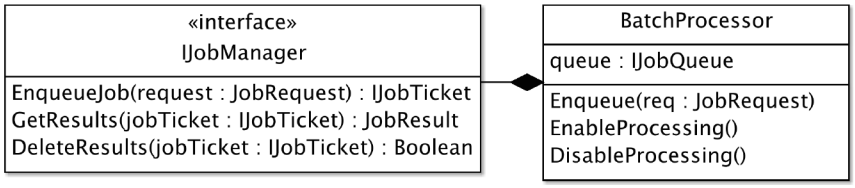

Figure 25. The *IJobManager* exposed to the public interface allows clients to enqueue their job requests, but delegates this enqueueing into an underlying *BatchProcessor* instance. The *BatchProcessor* retains the actual enqueueing and execution logic used by the processor.

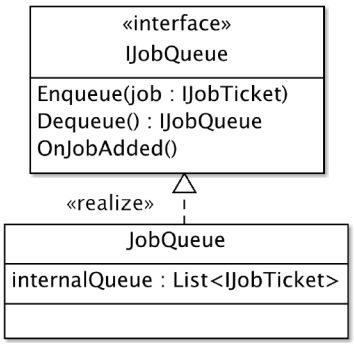

Figure 26. The *IJobQueue* is used to conceal the internals of the queue from other components. The underlying implementation in fact uses a *List*, providing thread-locking around adding or removing elements.

the job. The first of which represents the root of the underlying processing functions, known as the *BatchProcessor*. This object retains the *IJobQueue* currently in use, and is delegated to by the *IJobManager* when jobs are to be added. It accepts the job request, converts the information into a ticket, enqueues it and returns the ticket (figure 25). The conversion of the information encapsulated within the *JobRequest* does not actually occur until the job begins. When the job is dequeued, the background worker attempts to resolve the plugins associated with the *AlgorithmDefinition* objects within the pipeline. If it fails the job is scrapped and the client is signalled with an error, otherwise the process begins. The *BatchProcessor* does not perform this conversion itself, instead delegating to the previously defined *IPluginFactory* (figure 19). The *BatchProcessor* is provided with the current factory in its constructor, concealing knowledge of how the algorithms are created in addition to their source.

3.2.2. Job Queueing Client applications are able to establish a reference to the processor in order to dispatch jobs. As many applications could dispatch several jobs around the same time, or a single application requires processing of multiple batches, the design discussed the requirement of a capable queueing system (figure 2). The *IJobManager* exposes the ability to enqueue a job, in which the clients *JobRequest* is used as a parameter. The job manager itself does not retain this information, instead retaining an internal structure regarded as the *job queue*. The queue is represented as a very basic collection, only exposing mechanisms to enqueue or dequeue jobs. However it exposes an event notifying observers when elements are enqueued, providing a rudimentary alerting mechanism. There the potential for threading issues to occur when multiple applications access the queue simultaneously, or as a job is enqueued while an internal component begins to dequeue the next component. Therefore appropriate thread-locking needs to be implemented within the queue around dangerous areas, such as enqueueing and dequeueing jobs. Figure 26 depicts this structure. Hiding the true implementation of the queue structure behind the *IJobQueue* interface allows for dependency-injection based tests to be written, allowing simulating elements being added to the queue as part of a test. Additionally, the internal data structure used to represent the queue can be changed at will without breaking existing code, so long as the *JobQueue* continues

DOI: 10.1002/spe

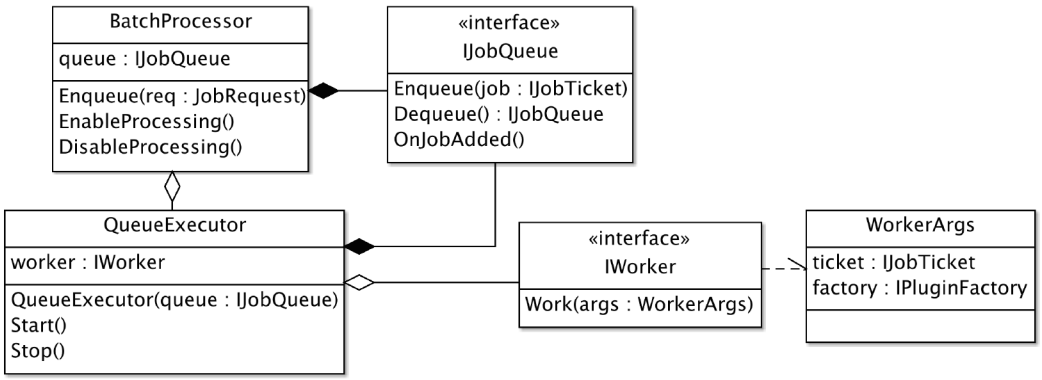

Figure 27. The *QueueWorker* dequeues jobs within the current *IJobQueue*, providing them to an *IWorker*. This removes knowledge of the queue from the worker, and increases the testability of the system by swapping the queue and worker implementations within the *QueueExecutor*.

to conform to the interface. The *BatchProcessor* retains the *IJobQueue* throughout the lifetime of the processor, however the dequeueing and execution process reside in two separate entities. During its construction, the *BatchProcessor* initialises an instance of the *QueueExecutor* class. The *QueueExecutor* uses an *IJobQueue* and listens for the job added event to be signalled. When it does, if it has yet to begin it spawns a separate thread used to dequeue the next job within the queue. This thread continues dequeueing elements until the queue is exhausted, where it simply terminates the thread. While the *QueueExecutor* dequeues the jobs, the object which processes them is further separated by interface. Upon dequeueing the next job, the *QueueExecutor* wraps the information within a parameter object along with the current plugin factory and provides it to an *IWorker*, which represents the object capable of processing jobs within the queue. While the level of decoupling in this situation seems overkill, it provides several benefits over heavily coupling the dequeueing and execution together. Namely, the testability of the system is heavily increased by separating the individual components as much as possible. *IWorker* implementations can be tested without a queue, and *QueueExecutor*s can have specific implementations of queues handed. Additionally, the source of the pending jobs is not known to the *IWorker* - it only possesses the knowledge needed to actually execute them and nothing more. This prevents locking the entire system to a queueing structure in the event a more efficient design is devised. In the current implementation, the *QueueExecutor* is provided with an *IWorker* implementation capable of fully executing the job described within the ticket. The *PipelineDefinition* within its request object contains the set of *AlgorithmDefinition*s chosen by the client, and can be handed to the *IPluginFactory* for conversion into *AlgorithmPlugin*s. Each input can then be ran through the processes before the output is generated. As the *IWorker* is provided with the *IJobTicket*, it is capable of firing the events through the accessible sink. This allows the client to be informed of many events as the job progresses, as the worker is aware of when it begins the job, as it progresses through each input, and when the job ends (or an error is occurred).

3.2.3. Event Management As execution of processing jobs is done synchronously, clients must be made aware through appropriate events when they reach completion. Further events can also be deployed to provide them with update information, for use within UI or other purposes. The client can attach to various events against the *IJobTicket* once they have received it from the job management system. However there are several additional points to note surrounding this:

1. Firing events through C# occurs on the same thread. Spawning a new thread to fire the event is a repeated process and should be kept in one location

2. If the events are placed within the interface declaration of the *IJobTicket*, the interface itself must change every time a new even is added or an existing event is removed

DOI: 10.1002/spe

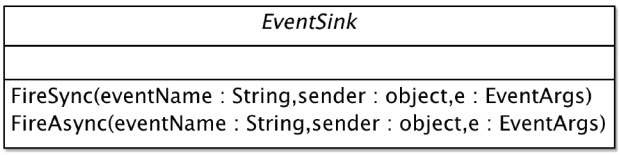

Figure 28. The *EventSink* class allows other classes than its subclasses to fire events against its instance. It also exposes the means to fire the event asynchronously, refactoring the common thread-spawning mechanism.

```
Type t = this.GetType();
BindingFlags flags = BindingFlags.Public | BindingFlags.NonPublic
    | BindingFlags.Instance;
EventInfo theEvent = t.GetEvent( name, flags );
```

Figure 29. The *Type* object allows locating an event against the type. The *EventSink* attempts to locate public or private events against instances of its subclass.

```
BindingFlags flags = BindingFlags.Instance |
    BindingFlags.NonPublic;
FieldInfo evField = GetType().GetField( ev.Name, flags );
Delegate theDelegate = (Delegate)evField.GetValue( this );
object[] parameters = new[] { sender, e };
theDelegate.Method.Invoke( theDelegate.Target, parameters );
```

Figure 30. On resolving the *EventInfo*, the *Delegate* methods observing the event can be resolved. These represent clients hooking into the particular event. The *Invoke* method on its *MethodInfo* member allows the method to be called as if the event was fired.

3. Additionally, events against classes can only be fired by the class itself.

Therefore, a common reusable class for firing events is required which allows the firing of events from other classes. This can be achieved by requesting the class fire a particular event by name, and use introspection on itself to locate this event. This class is regarded as an *EventSink* (figure 28), which exposes two methods for firing events. Calling either of the two fire methods within the *EventSink* class requires the name of the method, the sender, and an *EventArgs* object. The latter two are to satisfy the .NET standard for event method signatures. The name of the event is used to perform introspection against the current instance of the class (namely the base class). The *Type* class exposes a *GetEvent* method, which is used to locate the event identified by the name provided (figure 29). Once the *EventInfo* for the event is located, the event itself can then be invoked. Figure 30 demonstrates the manner in which the *Delegate*s hooked into the event can be resolved. The delegates represent methods observing the event - i.e., the client application listening for the event. This object may represent a single method, or it may be a *MulticastDelegate* representing several targets simultaneously. This satisfies the earlier points of firing events from other classes, and moving the common thread-spawning code somewhere common. The *IJobTicket* can also provide a subclass of *EventSink* and expose it on the interface. When an event is to be modified or added, only the *EventSink* subclass must be modified rather than the public interface itself. The design of the sink itself can be taken further. Consider the possibility the *EventSink* is used for processor-wide events across several applications. Rather than share one sink between the various clients, a container for multiple sinks can be exposed on the interface itself. Clients instantiate and add their sinks to the container and hook into the events of only their sink. This is depicted in figure 31. The *EventSinkContainer* class represents an aggregation of *EventSink*s, using a generic constraint on the *EventSink* class. It implements two methods from the *ISinkContainer* interface in order for client applications to add or remove their sinks, while internal implementations allow the processor to fire

DOI: 10.1002/spe

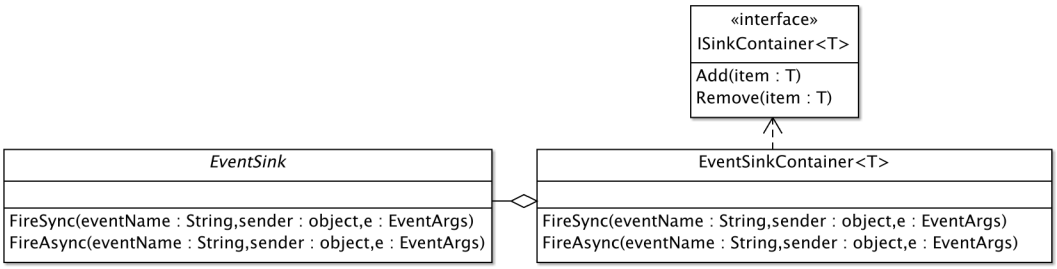

Figure 31. The *EventSinkContainer* allows multiple clients to add their own event sinks while still providing the asynchronous event firing mechanism. The *ISinkContainer* interface only lets clients add or remove sinks, and not fire events out of order.

the events across all sinks. In the case of the events from the *IJobTicket*, it exposes an *ISinkContainer* on its interface with a type constraint on *TicketSink*. The *TicketSink* contains the events relevant to the current job, such as the job beginning or ending, and so on. A client application can instantiate an instance of the *TicketSink*, add it to the container and hook into the relevant events. The processor will then fire the event asynchronously through a call to the actual *EventSinkContainer* on internal the *JobTicket* implementation.

3.3. Plugin Implementations

This section discusses the implementations of the various plugins provided with the product. While more plugins can be easily integrated into the system, several functions requested by the stakeholders have been provided as part of the product.

3.3.1. Common Image Processes As the pipeline system allows the user to compound multiple processes together, the first set of processes provided were decided to be relatively basic in their intent. As discussed within the literature review, the OpenCV library will be utilised to implement these common processes as it pro ides highly-optimised implementations of the algorithms; where it lacks in the specific algorithms, it also provides reusable components for easily developing algorithm implementations. A C# wrapper around OpenCV, EmguCV, was used to provide a direct method of calling these processes without re-development of a wrapper. The simple processes chosen for implementation include gamma correction, histogram equalisation, and various smoothing methods (figure 32). These were discussed within the literature review, and were chosen upon as the simple processes in which complex pipelines could be built during a stakeholder meeting. Implementing these plugins is as simple as extending the *AlgorithmPlugin* abstract class to provide the custom processing logic, copy the compiled DLL into the same directory as the processor, and start the software. The new plugins are then aggregated into the processor and exposed through the DIPS application. As these plugins are delegating the actual processing to EmguCV, their implementations are very basic. Figure 33 demonstrates the implementation of the histogram equalisation plugin, in which the processing is performed through a simple method call against one of the EmguCV classes. EmguCv provides the *Image* class to provide much easier access to properties or components of a digital image, in comparison to directly utilising the .NET *Image* class. This class itself provides all the common processes described within the literate review, and can be performed by simply calling the appropriate methods on the object. Once this is done, the *Output* property on the *AlgorithmPlugin* subclass is then set as the bitmap of the final image. The processor will use the value provided as the output from this process as the input to the next, without the knowledge that this plugin has utilised EmguCV rather than implementing the process manually. The remaining processes are just as simple to implement, wherein particular cases (such as smoothing) encapsulate the various smoothing methods within a separate class for testability reasons.

DOI: 10.1002/spe

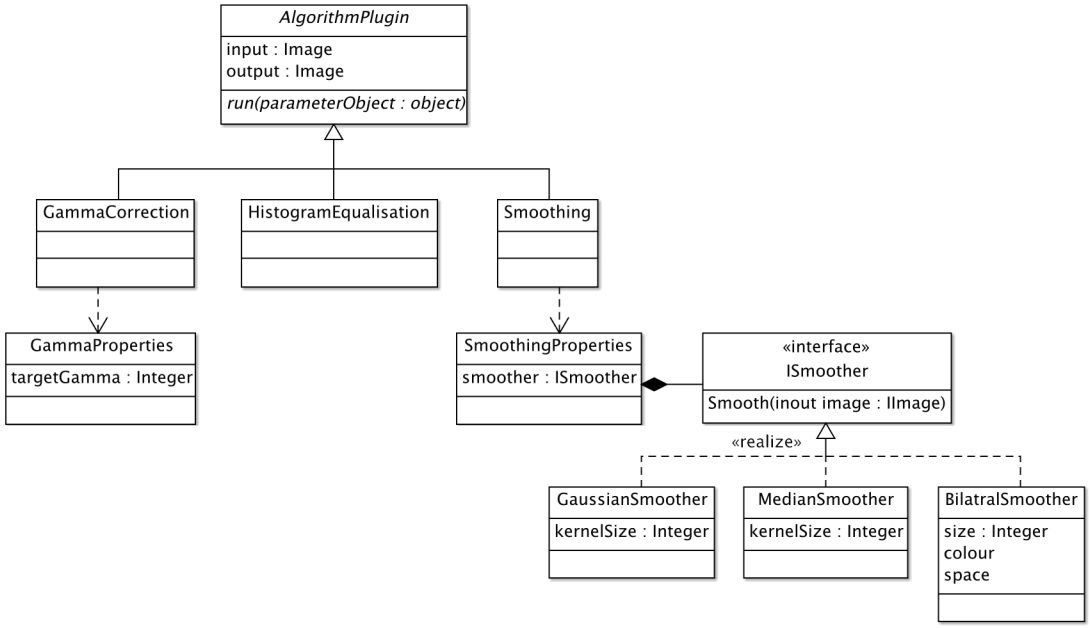

Figure 32. The common plugins depicted above are already implemented in EmguCV. The plugin system allows exposure of these processing methods as various plugins, to be compounded as part of a pipeline.

```
public override void Run( object parameterObject )
{
    Bitmap bmp = new Bitmap( Input );
    Image<Rgb, byte> img = new Image<Rgb, byte>( bmp );
    img._EqualizeHist();
    Output = img.Bitmap;
}
```

Figure 33. EmguCV reduces the load in implementing simple image processing algorithms, such as histogram equalisation

3.3.2. Matlab As part of the image processing solution, a feature of the application processing capabilities was to support the execution of Matlab scripts. This functionality can be provided as an independent plugin to further prove the extendability of the processor through the plugin system. The implementation of this plugin is split into two components, including a wrapper around the Matlab COM interface and the actual *AlgorithmPlugin* implementation.

Matlab COM Wrapper Matlab exposes a single object, the *MLAppClass*, through COM (Component Object Model) allowing other applications to take advantage of its features. The *MLAppClass* object allows setting and getting of values from the Matlab workspace, execution of commands as Matlab scripts, and more. To isolate the separate logic layers, the direct communication with the COM interface is separated into a set of classes utilised by the plugin (figure 34). The various classes depicted above are explained in detail in the following section.

MLAppClass

The *MLAppClass* is the COM object provided by Matlab, consisting of the set of methods it exposes. It possesses a variety of methods for getting or settings objects on the workspace, which have been omitted for clarity. The main methods include the *Execute* method, which accepts a single line of Matlab script and executes it against the engine. *GetWorkspaceData* and *SetWorkspaceData* are self-explanatory, however it is important to note complex types

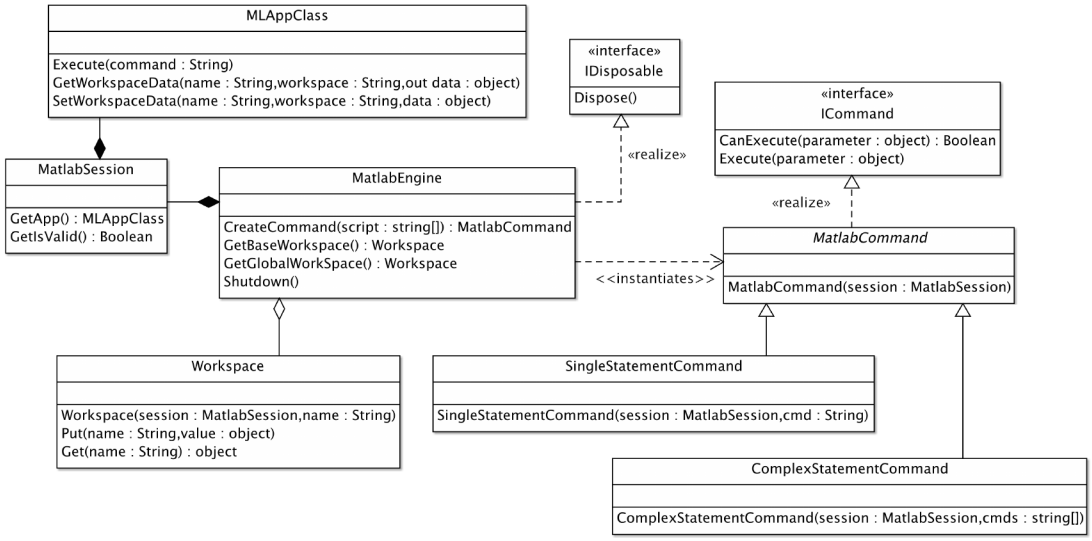

Figure 34. Due to the complexities involved in communication, the COM object provided by Matlab is not directly communicated with. The set of wrapper classes manage the interface, with the plugin utilising the *MatlabEngine* when it begins to run.

cannot be provided to Matlab. Values transmitted in these methods but be primitives, with the exception of strings.

MatlabEngine

The primary class utilised by the plugin is the *MatlabEngine*, which initially creates the instance of the COM object. The plugin can use the *MatlabEngine* to create script commands or access the available workspaces. Once it has finished using the engine, it can call *Shutdown* to terminate the Matlab session.

MatlabSession

In order to share the *MLAppClass* instance across the various components of the wrapper, the *MatlabSession* class contains the current engine currently in use. It is originally instantiated within the *MatlabEngine* and provided to the various components as they are created, allowing them to also assert whether the session is valid (i.e. Matlab is running)

Workspace

The *Workspace*s exposed by the *MatlabEngine* make it simpler for the plugin to get or set values against the Matlab workspaces. The *MatlabEngine* instantiates the 'Base' and 'Global' workspaces on startup, in which calls to *Put* or *Get* automatically provide the workspace name.

MatlabCommand

To execute Matlab scripts against the engine, the *MatlabCommand* is instantiated by the *MatlabEngine* and provided to the client. This command retains the current *MatlabSession*, allowing re-use of the command so long as the session remains active. Depending on the amount of lines of script provided, different implementations of this class are provided.

SingleStatementCommand

One implementation of the *MatlabCommand* is the *SingleStatementCommand*, allowing the execution of a single line of Matlab script. Calls to *Execute* simply execute the one line of script it had previously been provided with within its constructor.

ComplexStatementCommand

Another implementation of the *MatlabCommand* is the *ComplexStatementCommand*, which

DOI: 10.1002/spe

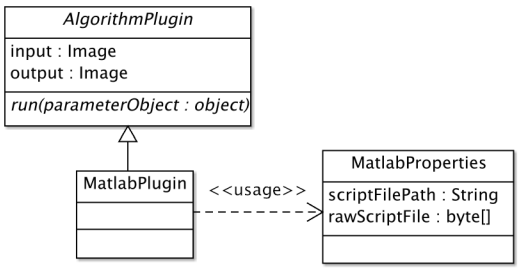

Figure 35. The *MatlabPlugin* represents an algorithm plugin capable of running a Matlab script to process the images. It's relevant parameter object retains a raw copy of the script for persistence purposes.

```
input = imread(dipsinput);

...

imwrite(out, 'Output.bmp');
dipsoutput = 'Output.bmp';
```

Figure 36. In order for the DIPS plugin to communicate with the Matlab script, some minor changes must be made to it. The plugin provides the path to the input image within a *dipsinput* variable, and requires a path to the output within a *dipsoutput* variable.

accepts a number of commands for sequential execution. The other in which the strings are provided represents the order in which they are executed.

Separating the plugin logic into separate components, including the Matlab COM wrapper, helps in breaking down the system into more manageable chunks. It also makes it more feasible to write unit tests against the Matlab COM wrapper separately from the Matlab plugin itself.

Implementation In order for Matlab scripts to actually run within a pipeline, the associated *AlgorithmPlugin* needs to be developed. The plugin itself will use the script provided within the associated parameter object to process the current image within the pipeline. Figure 35 depicts the UML for the Matlab plugin, in which the *MatlabProperties* parameter object retains a raw copy of the original script. This is used when saving the algorithm to Xml rather than retaining the original copy of the script. The *MatlabPlugin* uses the COM wrapper previously designed to communicate with Matlab. It instantiates a new instance of the *MatlabEngine* class when the *Run* method is invoked and constructs a command using the information within the script. As complex types such as images cannot be directly provided to Matlab through the COM interface, the images must be temporarily saved to disc. The path to the image is then provided to Matlab in a 'dipsinput' value on the workspace, in which the script must make use of this path instead. The script loads the image using the *imread* Matlab function, continues processing as normal, then outputs the result of the processing at the end of the script with a call to *imwrite*. It then sets the path of the output to a variable on the workspace named 'dipsoutput', which the plugin locates and loads the image from once the script has terminated. Due to this manner of saving and loading images, scripts must be adjusted accordingly to allow them to function as part of the pipeline. However these changes are fairly minor, only requiring the script to load an image from a file path already provided by the plugin.

3.4. Summary

The provided implementation of the DIPS processing module has satisfied the requirements produced by the design, which in turn were based off the revised aims and objectives of the processor. This ensures the DIPS application is well-equipped with the processing functionality

DOI: 10.1002/spe

required by the user to perform the MRI pre-processing functions. The application of the various object-oriented principles and processed outlined in the design have allowed the processing module to remain very testable. The primary reason for this is the application of dependency injection, providing the means to devise automated tests through the use of mock instances of classes. Additionally, both internal and external components are hardly coupled to one another and instead rely predominately on abstractions. Separating the implementation of image processing algorithms into plugins ultimately proved beneficial during the implementation cycles. Once the core functions of the processor were constructed and tested, the processes could be incorporated separately without the fear of breaking existing functionality. This especially paid off during the implementation of the Matlab plugin, which was requested later on during the development lifecycle.

4. TESTING

4.1. Unit Testing

As the processing module is intended to be independently only as a service or background worker, no specific UI has been designed for it. Additionally, not all functionalities can be tested through an interface as validation constraints would restrict the ability to assert certain cases.

Instead, individual unit tests have been designed for each class used by the implementation of the processor. As many public methods as possible have associated unit tests, in order to provide as much coverage as possible. These tests ensure the individual methods and classes are functioning properly, as opposed to the system as a whole.

The unit tests designed for the system were re-run frequently during development in order to ensure any pulled commits from the shared repository did not break existing code. In contrast to manually testing the system, this drastically saved time and effort.

4.2. Integration Testing

During the development of the project, infrequent phases of integration testing occurred to ensure the processor and the primary application were continuing to function normally. These tests were performed less formally than unit or validation tests; any bugs raised during this time were noted and assigned to the relevant team member to investigate and fix the problem.

4.3. User Acceptance

Much less frequently than the aforementioned forms of testing, several runs of validation by the stakeholders were performed against stable prototypes of the end product. These occurred once or twice per quarter, allowing them to ensure the product was going in the right direction and giving them an opportunity to request changes or features.

5. CONCLUSION

The aim of this DIPS project was to develop an image processing module capable of performing batch processing on behalf of client applications. This goal was achieved, and taken further by providing a mechanism to easily provide new methods of processing images by implementing plugins to the processor representing individual image processes. While the entirety of the processing logic required by the design has been fully implemented and tested, there continues to remain scope for building new algorithm plugins capable for use by clients. Further required processes can also be implemented through the use of the plugin architecture provided by the processor. Future work involves futher development of the DIPS to fully support running as a service. The framework for this is in place, and the capability to connect to another instance of the processing service through TCP/IP is an achievable possibility. This would allow the processing service to reside on a separate computer, which could be host to a much faster hardware. For

DOI: 10.1002/spe

instance, slower clients could dispatch the jobs to the alternate machine, making best use of its capabilities.

REFERENCES

1. Chang PL, Teng WG. 2007. *Exploiting the self-organizing map for medical image segmentation* in Computer-Based Medical Systems, Twentieth IEEE International Symposium on, pp. 281288.
2. Zhou ZJ,Wu H. 2010. *Digital Image Processing: Oart 1*. Ventus Publishing ApS.
3. Lo WY, Puchalski SM. 2008. *Digital image processing* Veterinary Radiology and Ultrasound, vol. 49, pp. S42S47 [Online]. Available: http://dx.doi.org/10.1111/j.1740-8261.2007.00333.x
4. Zhang Y. 2009. *Image processing using spatial transform*. in Image Analysis and Signal Processing, IASP 2009. International Conference on, pp. 282285.
5. Larkins DB, Harvey W. 2010. *Introductory computational science using fMATLABg and image processing*.Procedia Computer Science, vol. 1, no. 1, pp. 913 919, fICCSg 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050910001018
6. Culjak I, Abram D, Pribanic T, Dzapo H, and Cifrek M. 2012.*A brief introduction to opencv*. in MIPRO, 2012 Proceedings of the 35th International Convention, pp. 17251730.
7. Matuska S, Hudec R, and Benco M.2012. *The comparison of cpu time consumption for image processing algorithm in matlab and opencv*. in ELEKTRO, May 2012, pp. 7578.
8. Mankoff J, Blevis E, Borning B, Friedman A, Fussell SR,Hasbrouck J, Woodruff A, and Sengers P. 2007. *Environmental sustainability and interaction*. pp. 21212124.